

Selection of Methodology for Representing a Domain Theory

Automatic Software Code Generator

Deliverable A1

Nicolas F. Rouquette

DS1 Background

Deep-Space One is the first JPL spacecraft to use large-scale automatic software code-generation techniques on a large scale to produce the fault-protection software of the spacecraft.¹ The code-generation techniques used on DS1 consist in separates two types of design knowledge. One kind of knowledge describes artifacts about the DS1 spacecraft and the fault protection requirements, we generically refer to it as domain knowledge; examples include specific fault-protection designs such as protection against attitude control error, command loss, battery low-state of charge, inertial reference loss, etc. The second kind of knowledge describes the generic software architecture principles and designs selected to support any and all fault protection designs such as initialization, storage of parameters in non-volatile memory (EEPROM), ground parameter update commands, execution of a fault response, detection of a fault condition, etc... The task of the automatic code-generation process then consists in applying the domain-independent principles of software architecture and organization to the domain-specific requirements of fault-protection design and capability.

To meet the very tight schedule of the DS1 project, the code-generation mechanisms were deliberately kept flexible to accommodate new requirements and constraints as the project evolved. Domain-independent software architecture designs and principles were described in terms of *software templates*. Without code-generation, the fault-protection software would be embodied in a set of files, each written by hand. The idea of a software template is to find similarities among all of the product's software files at the level of software organization and architecture. Files with similar organization and architecture are then grouped and their domain-specific aspects are generalized into domain description properties (e.g., replace properties such as 'red' and 'green', with a generic property 'color'). Instead of doing this generalization *after-the-fact*, the challenge of DS1 fault-protection was to design the generic templates *up-front*, apply them to the on-going current version of the domain-specific design and requirements and see how well the generated code meets the product and software requirements.

Code Generation Complexity

Throughout the project, a substantial number of modifications were made to the code generator per-se. Each modification made can be traced to one of two issues: either a discrepancy with respect to new or changed requirements about software architecture and organization; or new end-product requirements necessitating taking into account new design information. The compound effect of such changes resulted in a very complex code-generator, one that violates the original idea of decoupling domain-specific fault-protection knowledge from domain-independent architectural & organizational knowledge.

Analyzing how support for telemetry was added to the code generator is a good illustrative example of how complexity built-up throughout the project. The fault-protection software includes a collection of objects known as *monitors*. The purpose of each monitor is to passively observe a stream of real-time data and to make a decision when a symptom of some anomaly is present in the data. To do so, a monitor maintains a number of state variables that are typically used to characterize properties about the data stream history up to the present. Initially, telemetry was considered to be a simple, passive observational account of monitor state variables. When support for telemetry was finally introduced in the code, new requirements suddenly crept in through analysis. In nominal cruise, the spacecraft is scheduled to spend up to 6 days of unattended thrust. To provide visibility into abnormal events that might occur during that time, the design of the

¹ The DS1 flight software comprises several large modules for attitude control, autonomous navigation, flight systems control (including the ion-propulsion drive) and system-level fault protection. The last module is the largest of all in terms of the binary code stored on the flight computer.

monitor telemetry was expanded to include water marks and several statistics measures defined to summarize the outcome of decisions made regarding the degree of abnormality in the raw data. Although the concept of telemetry is simple, the code generator had to be expanded to make several software inferences including change data propagation (e.g., telemetry is updated *after* a variable is updated, wherever this update may occur), and type inference (e.g., what are the possible outcomes of a decision-making function). The net result is that defining something like telemetry requires additional derived knowledge about the software itself and how it is built and this is clearly beyond the scope of simple syntactic replacement of generic property references by domain-specific values.

Capturing Domain Information in a Model Database

We believe that the notion of separating domain-specific design knowledge and domain-independent knowledge about software architecture and organization is still a sound idea, although it is now clear that carrying it through is a technically challenging task. The problem about adding telemetry support in DS1 demonstrates the need for making *inferences* during code generation. Many other problems pertaining to software integration and maintenance can be traced to *dependency management*. Any generated file has a dependency on all the sources of information used in the code-generation process; therefore, we need to know when a change made somewhere potentially affects a generated product and therefore warrants analysis and possibly re-generation.

Aspects of dependency management and knowledge compilation indicate the need for a database-management approach to domain-specific design information. This approach entails modeling the domain into a database schema and populating the database with actual instances of domain objects. Once all relevant domain information is captured in a database, the problem of dependency management can be readily addressed by including in the database the appropriate dependency information. From a database standpoint, the code-generation process involves querying the database for the attributes and properties of the domain necessary to perform the necessary generic property replacement in a software template. Similarly, the inference processes mentioned above have database counterparts as a database query feeding an information engine whose results are stored back in the database for inferences or code-generation processing. This latter aspect of information processing implies the definition of inference rules to define what domain knowledge will be combined and how this combination will take place to derive inferences about the domain. This means that the domain is not only modeled in terms of a static schema of properties and relation but also in terms of a theory about relationships among the elements of the domain.

This database-driven strategy for code-generation and inferencing appears sufficiently sound to select a modeling tool to support all of the above-mentioned processes. This choice is important since our customer, the X2000 initiative at JPL, is also considering the use of models, at least, for review purposes. There are a large number of research efforts to define modeling paradigms and domain theories for the purpose of mechanizing inferences, dependency management and code-generation (e.g., from Artificial Intelligence research) and for the purpose of code generation (e.g., from Software Engineering research). So make a relevant research product to our customer and potential customers, we chose to settle on a modeling approach based on the Unified Modeling Language (UML) because of its unprecedented broad industry, research, and tool support. Rational Corp. is one of the champions of UML and has committed a lot of resources in developing and supporting tools for applying UML in the context of modeling, code generation, reverse engineering, round-trip engineering processes, requirements engineering, project planning, and testing. Consequently, we chose to use Rational Rose and its associated tools to provide the modeling support we need to carry out this project.

Modeling a Domain Theory

Rational Rose provides a flexible modeling environment, where we define the domain-independent software architecture and organization on one hand and the domain-specific schemata of information on the other. In the course of the project, we will define what aspects of the domain theory can be captured in this tool and which others require extending the tool. As it stands, Rational Rose supports modeling of *use cases*, scenarios describing what the modeled artifact is supposed to do and *logical categories*, detailed descriptions of the structure and organization of how the artifact will achieved what it is supposed and expected to do.

We plan to apply this modeling framework in the following manner:

- describe in terms of use cases what mechanisms the software architecture is supposed to provide
- describe in terms of logical design packages of class information what domain-specific information is needed to characterize specific fault-protection requirements, knowledge and design
- describe in terms of logical design packages the processes whereby domain-specific information is applied according to the use cases to generate domain-specific fault-protection software, documents, reports, etc..

Things to come

The next delivery will address a case-study of a domain-theory describing a subset of the DS1 fault-protection design.

Contributors for this deliverable

Nicolas Rouquette
Julia Dunphy